PERFORMANCE OF GENETIC ALGORITHMS FOR DATA CLASSIFICATION

by
Matthew E. Stine

A thesis submitted to the faculty of the University of Mississippi in partial fulfillment of the requirements of the McDonnell-Barksdale Honors College.

Oxford
May 2001

Approved by:

_____
Advisor: Professor Dawn Wilkins

_____
Reader: Professor H. Conrad Cunningham

_____
Reader: Professor James Vaughan

To Wendy, my motivation.

ACKNOWLEDGEMENTS

I thank Dr. Dawn Wilkins of the Department of Computer and Information Science, who introduced me to data mining, absorbs my endless "sob-stories," provides me with the monetary support to keep food on the table, and gave me a free ride to the CAMDA 2001 conference at Duke University, at which I first heard these words: "genetic algorithm."

ABSTRACT

MATTHEW E. STINE: Performance of Genetic Algorithms for Data Classification
(Under the direction of Dawn Wilkins)

In today's world, the amount of raw data archived across multiple distinct domains is growing at an exponential rate. "Data Mining" is a continuously evolving family of processes by which individuals extract useful information from these data. Classification is one of these processes, and is the construction of varying types of descriptive models from labeled data objects, for the purpose of predicting the label of those objects with unknown labels. The construction of these models is often adversely affected by the presence of incorrect values or outlier values within the data, a phenomenon known as noise. The original motivation of this research was to test the performance of the binary genetic algorithm, one of a multitude of algorithms used for model construction, in the presence of data with varying percentages of noise. However, in the course of experimentation, several issues arose concerning the effectiveness of the binary genetic algorithm as a classifier. Specifically, the chosen method for encoding classification hypotheses demonstrated limited scalability. Furthermore, the chosen method for encoding continuous and nominally valued data attributes was discovered to be unreasonably strict, leading to poor performance. Further research should be undergone to investigate a more reasonable encoding method. However, the algorithm performed favorably on purely categorical data with a relatively moderate number of small-domained dimensions. Upon injecting

varying percentages of noise into these data, the algorithm exhibited a slow, steady decent in classification accuracy. These results lead to the conclusion that the binary genetic algorithm should not be discounted as a possible answer to the question of data classification, especially for data sets with the above characteristics, and further research could reveal hypothesis encoding strategies that will result in improved scalability.

Contents

1 Background **1**

1.1 Introduction 1

1.2 Classification 4

1.3 The Binary Genetic Algorithm 6

1.4 Noisy Data 10

2 Design and Methods **12**

2.1 Implementation Environment 12

2.2 *EvoLearner* Design Specification 14

2.2.1 Input File Structure 14

2.2.2 User Input and Data Preprocessing 15

2.2.3 Classifier Discovery 19

2.2.4 Output Results and Cleanup 22

2.2.5 Fitness Function 25

2.2.6 Sorter Function 26

2.3 *NoiseMaker* Design Specification 27

2.3.1 User Input 28

2.3.2 Noise Addition and Output 29
2.4 Experimental Design 32
2.4.1 Algorithmic Parameters 32
2.4.2 Test Data Sets 33
2.4.3 Experiments 33

3 Results **35**

3.1 House Votes 1984 35
3.2 Wisconsin Breast Cancer 39
3.3 *Agaricus lepiota* 40
3.4 Monks 41

4 Conclusions and Future Challenges **44**

4.1 Conclusions 44
4.2 Future Challenges 50

List of Figures and Tables

Figure 1 EvoLearner Performance
Noise in Multiple Attributes
House Votes 1984 36

Figure 2 EvoLearner vs. C4.5
Noise in Multiple Attributes
House Votes 1984 37

Figure 3 EvoLearner Performance
Noise in Single Attribute – Physician Fee Freeze
House Votes 1984 38

Figure 4 EvoLearner vs. C4.5
Noise in Single Attribute – Physician Fee Freeze
House Votes 1984 38

Figure 5 EvoLearner vs. C4.5
Noise in Multiple Attributes
Monks 42

Figure 6 EvoLearner vs. C4.5
Noise in Single Attribute – Jacket Color
Monks 43

Figure 7 Growth of Hypothesis Space for Binary Encoding 46

Table 1 EvoLearner Parameters 32

Table 2 EvoLearner – Performance on Wisconsin Breast Cancer 39

Table 3 EvoLearner – Performance on *Agaricus lepiota* 40

Chapter 1

Background

1.1 Introduction

In today's world, the amount of raw data archived across multiple distinct domains is growing at an exponential rate. With each scan of the popular discount cards now used by nearly every supermarket in America, weekly grocery purchases are documented and cross-referenced against the demographical data provided by registered customers. Research hospitals generate gigabytes of genetic data on a daily basis as they catalogue the nucleotide base pair sequences contained within the cells of those stricken with catastrophic diseases. As you read this thesis, expert systems are rejecting and approving credit applicants, with their decisions based solely on answers to seemingly unrelated questions. Just as programming has become increasingly "data-driven" with the popularization of object-oriented programming, so our world is also becoming "data-driven." We certainly appear to have mastered the task of data collection and storage. Now arises these questions: What do we do with the

data? Can they teach us anything? Is there some method by which we can extract useful information from these data? These are the questions now being studied in the still young field of "data mining."

Suppose that we wish to take a collection of data objects that have known labels and use them to construct a model for the purpose of labeling new data objects as they are recorded. This task is a loose definition of the subsection of data mining known as *classification*. Now further suppose that this collection of data objects has several attributes that, by virtue of an inefficient data collection process, contain varying percentages of incorrect values. This is one phenomenon that is described within the data mining community as *noise*. It was the original purpose of this research to evaluate the performance of the binary genetic algorithm, an optimization algorithm modeling a subset of biological evolution, in classifying data containing varying percentages of noise. For these purposes, two applications were designed and developed:

1. *NoiseMaker*, an application for the purpose of injecting precise amounts of random noise into data otherwise known to be clean;

2. *EvoLearner*, a generic classification rule generator powered by a binary genetic algorithm.

These applications were used to conduct experimental trials using various data sets, injecting them with varying percentages and breeds of noise. As experiments were conducted, the results pressured the "evolution" of this study into an evaluation of the binary genetic algorithm's effectiveness simply as a generic classification tool,

with noise or without it. Several important observations were made, resulting in the hypothesis that the binary genetic algorithm can be a useful and robust tool for data classification, but further research into methods for candidate hypothesis encoding must be pursued.

1.2 Classification

Classification is the process by which a set of models is derived that distinguishes data classes, for the purpose of using the models to predict the class of unlabeled data objects. The model is constructed through analysis of a *training set*, a set of labeled data objects. These models may be presented in a variety of forms, including but not restricted to:

- Classification ($if \ldots then$) rules
- Decision trees
- Mathematical formulae
- Neural networks

The basic process is as follows. A classification algorithm analyzes the training set, which consists of data records described by attributes. One of these attributes is the *class label attribute*. The result of this analysis is the production of some model, which may or may not be outwardly interpretable. For instance, the learned weights and biases of a neural network model make little sense to a human being, even though they are excellent classification tools. This step is often referred to as *supervised learning*, because each training record is labeled for the algorithm's benefit. One can contrast this procedure with *unsupervised learning* tasks, such as clustering, in which no training records are labeled, and it is the algorithm's responsibility to define the data classes.

Following a model's construction, it may be used for classification. However, first the model's accuracy must be estimated. There are a number of accuracy tests currently in use, with the simplest being the *holdout method*. This technique utilizes an additional *testing set* for estimating the model's accuracy. Before model construction, a specified percentage (typically $\frac{1}{3}$) of the data is "held out" for testing purposes, with the remaining records used for model construction. The actual class label for each testing record is compared with the model's predicted class label for that record. The percentage of testing records correctly classified by the model defines the model's *accuracy*. The holdout method is used because an accuracy measure based solely on the training set could be overly optimistic. Classification models often *overfit* the training data by incorporating features and anomalies of the training set that may not be present in the overall data population [HK01].

Probably the most well-known method for data classification is the *decision tree*. Decision trees are graph-like structures, with each node representing an attribute value test. Edges represent an outcome of the test, and "tree leaves" represent classes. Classification rules can easily be extracted from decision trees simply by tracing paths from the root to an individual leaf. One of the more common versions of the decision tree algorithm is J. R. Quinlan's C4.5, the successor to his own well-known ID3 [Qui93]. Various implementations of C4.5 are widely-available. For the purposes of this research, an implementation included with the Waikato Environment for Knowledge Analysis (WEKA) [WF00] will be used as a basis of comparison for the binary genetic algorithm.

1.3 The Binary Genetic Algorithm

Genetic algorithms were developed by John Holland and his students and colleagues at the University of Michigan during the 1960's and 1970's. They are an abstraction of biological evolution, and were first described in his 1975 book *Adaptation in Natural and Artificial Systems*. At this point it seems useful to introduce some of the biological terminology that will be used in the description of the algorithm.

All living organisms consist of cells, each cell containing the same set of one or more *chromosomes*. A chromosome is a single molecule of DNA, and serves to encode the information necessary to carry out the metabolic functions of the organism. Chromosomes can be divided into functional units called *genes*. Some genes encode proteins, while others encode cellular machinery used to control the expression of the protein-encoders. Informally, we can think of the protein-encoding genes as encoding single "traits," such as eye and hair color. Genes that encode different results for a particular trait are called *alleles*. We can say that the eye color gene has alleles for blue eyes, green eyes, etc...

Organisms whose chromosomes are arranged in pairs are called *diploid*; those whose chromosomes are not arranged in pairs are *haploid*. Most sexually reproducing species in nature are diploid, although several haploid sexually reproducing species exist. During sexual reproduction, *recombination* (*crossover*) occurs. Genes are exchanged between the chromosome pairs to form a single haploid *gamete*. Gametes from two parents are then brought together to create a new diploid individual. In haploid organisms, genes are exchanged between the haploid chromosomes of the

two parents to create a new haploid individual.

Offspring are subject to *mutation*, a type of which occurs when single nucleotide base pairs (the building blocks of DNA) are incorrectly substituted. These mutations are often the result of copying errors during the transmission of genetic information from parent to offspring.

Recombination and mutation serve to create genetic diversity in the population, and it is this genetic diversity that serves to fuel the engine of evolution, which is founded upon the idea of "survival of the fittest." Here *fitness* is defined as the probability that an organism will survive long enough to reproduce. The processes of recombination and mutation can create individuals with a genetic advantage, thereby increasing their fitnesses; however, it can also create individuals with a genetic disadvantage, and these individuals often die far too quickly to pass on their genetic information [Mit96].

For the binary genetic algorithm, we will be concerned with populations consisting of haploid individuals bearing a single chromosome. In fact, this chromosome will actually encode a candidate classification rule (hypothesis) for a particular data set. The chromosome will consist of bit strings (i.e. strings of 1's and 0's). The genes are either single bits or blocks of bits, and encode the values of individual attributes. The conjunction of these values will imply the label (which is encoded by the first gene on the chromosome) that is given to data records fitting the rule encoded by the chromosome.

The algorithm generates a random population of individuals (henceforth referred to simply as chromosomes). These chromosomes are then evaluated by a fitness

function, which for this incarnation of the algorithm measures the support and confidence percentages for the encoded rules with respect to the training data. Letting the attribute values specified by the rule equal A and the class label specified by the rule equal B, support is given by [HK01]:

$$support = P(A \cup B) \tag{1.1}$$

and confidence by:

$$confidence = P(B \mid A) \tag{1.2}$$

The fitness is then calculated by:

$$fitness = support \times confidence \tag{1.3}$$

The n fittest chromosomes are kept for reproduction, and then the population is returned to its original size by using some pairing algorithm to mate the fit chromosomes and produce offspring. A crossover operator is then applied to each mating pair. A random point in the bit string is chosen, and all of the bits before that point in the first parent, and those after that point in the second parent are passed on to the first child. The second child receives the remaining bits in their correct order. Mutation occurs by "flipping" individual bits in the population according to some specified mutation rate.

This process is carried out iteratively until either a specified maximum number of generations has been carried out, or until some desired fitness threshold is achieved.

At this point the best chromosome and its encoded rule are output [HH98, Mit96, Mit97].

The theory behind why the algorithm works is actually quite complicated, and is beyond the scope of this project. Those seeking this information will find an excellent synopsis in Chapter 4 of [Mit96]. However, it is necessary to provide some justification for the use of this algorithm in seeking the optimal classification rule for some domain. The key is once again found in the idea of "survival of the fittest." The fitness function assigns some score to a chromosome based on how well its encoded rule classifies the data records. It is assumed that those rules that do a "good" job classifying the data contain some of the elements that make up the "perfect" rule. Thus we want only the chromosomes encoding "good" rules to survive and reproduce, which is exactly what happens. We then couple this idea with the crossover operation, which allows the elements that make up "good" rules to be recombined into possibly "better" rules. Finally, adding the mutation operation may allow those rules that may be only slightly "off" to become "better" rules. Repeating this process for several generations will hopefully cause the population to converge on that one "perfect" rule, if it in fact exists.

1.4 Noisy Data

In today's world of data mining, one of the significant problems is the presence of noise within data sets. *Noise* is defined to be found in values that contain errors, or in outliers, values that deviate far from the expected [HK01]. There are several reasons that data sets could contain noisy values. For instance, consider an automated data collection process, where assorted instruments measure the conditions during and/or results of some manufacturing process. Now imagine that one of the sensors is somehow jarred, and its calibration ruined. If this malfunction were to go unnoticed, large amounts of data relating to that particular sensor would now be corrupt. Here we see a data set where one individual attribute is always incorrect, with the remainder of the data assumed to be accurate.

Another possible source of noise is derived from the fact that even in today's highly automated world, a great deal of data entry is still done by humans. As we all well know and have experienced, humans often make mistakes when entering data into a system. To quote a personal example, last year my checking account statements suddenly stopped arriving in the mail, even though I had forwarded all of my mail to my new address. When I called the bank to change my address, I was reasonably asked to verify my old address. However, a problem arose: I could not! I rattled off every address that I had lived at since I had opened the account, and none were the address listed. After verifying some other information, I learned that the address stored in their computer system was one at which I had never lived; nor did I know who did! I can only attribute this event to some human data entry error.

A third possible source of noise is corrupt data transmission. As data flows around the world through telephone lines and the countless routers, hubs, firewalls, and proxies of the Internet, it is nearly impossible for error free transmission to always occur. Both this source and the previous one lead to data sets where any number of values across any number of attributes may be incorrect.

This list of noise sources is by no means exhaustive, but is meant to illustrate that noisy data sets are indeed a real problem. There is currently extensive research into the removal of noise, or data smoothing. Also, researchers are continually trying to improve old and design new algorithms that can better deal with data sets that still contain noise. It was the original purpose of this project is to see just how well the binary genetic algorithm described above could find classifiers for data sets of the two types described above: those with errors in only one attribute, and those with errors in many attributes.

Chapter 2

Design and Methods

2.1 Implementation Environment

Given that the primary operations performed by both the desired applications will be input/output (I/O) and array/string manipulation, it seems logical to choose an implementation language that provides powerful and efficient methods to perform those tasks. Further, it is desirable that the applications, which will be used in further research beyond the scope of this study, be executable on a wide range of platforms. Larger data sets, both in number of records and attributes, may require the added computational and storage power of multiprocessor servers, which normally run some variation of the UNIX operating system; however, these applications are also intended for use on personal workstations when data sets are small enough for their limited resources. These systems normally run some variation of Microsoft Windows. Therefore, the implementation language should also be easily portable between those operating systems. Finally, the language should provide some facilities

for implementing an intuitive graphical user interface (GUI).

The Practical Extraction and Report Language (Perl), coupled with the Tk interface package, adequately fulfills each of these three requirements. Complicated manipulation of array-based data structures is accomplished with very little developer effort. File I/O is trivial, and delimited data files are easily loaded into two-dimensional (2D) arrays using Perl's powerful pattern matching operations. Perl is an interpreted language, and free Perl interpreters are readily available for both PC/Windows and UNIX platforms. Scripts will run unedited under either interpreter. The Tk interface provides easy-to-use object oriented facilities for creating graphical user interfaces (GUI's) and is also fully portable between PC/Windows and UNIX platforms. Therefore, Perl/Tk will be the implementation language for this project.

For developmental hardware purposes I have chosen an 800 mHz–Pentium III workstation equipped with 256 MB of RAM, dual-booting Microsoft Windows NT Workstation 4.0 and Red Hat Linux 7.0. The ActiveState binary distribution of the Perl interpreter is installed on the NT partition, coupled with CPAN's port of the Tk interface package, which is compiled with Microsoft Visual C++. The Linux partition comes with the Perl interpreter preinstalled, and CPAN's port of the Tk interface package was added and compiled using the GNU C compiler.

Successful implementation and testing in this hardware/software environment should result in fully portable applications that are capable of handling both large and small classification data sets.

2.2 *EvoLearner* Design Specification

EvoLearner will be a generic tool for discovering classification rules for data sets, powered by a binary genetic algorithm. It will "generic" in the sense that it should be useable on any classification training set with a binary-valued (i.e. yes/no, republican/democrat, etc...) class label. In the future I hope to extend it to accept data sets having more than two classes. *EvoLearner* will be able to process three types of attributes:

1. *Nominal-Valued* Attributes, which are drawn from the set of Integers, $\mathcal{Z}$.
2. *Continuous-Valued* Attributes, which are drawn from the set of Real numbers, $\mathcal{R}$.
3. *Categorical* Attributes, which consist of a finite set of user-specified values.

2.2.1 Input File Structure

The user will specify the structure and content of his *data file* through the use of a *schema file*. The two files should share the same name, with extensions of `.dat` and `.schema` respectively. Schema files should follow this format:

```
LabelDescriptor LBL     Class1  Class2
Attribute1Descriptor    INT
Attribute2Descriptor    REAL
Attribute3Descriptor    CAT     Value1  Value2  Value3  Value4
```

The file should be tab-delimited. The first column should contain the descriptor for each attribute (i.e. Age, Outlook, etc...). The second column indicates the data

type of each attribute, with `LBL` denoting the class label, `INT` denoting nominal-valued attributes, `REAL` denoting continuous-valued attributes, and `CAT` denoting categorical attributes. Following a `LBL` or `CAT` should be a tab-delimited list of possible values.

The data file should be comma-delimited, with the rows being individual records and each column corresponding to a row in the schema file. In both files, the first attribute **should always be the class label!**

EvoLearner's execution can be divided into three distinct steps:

1. User Input and Data Preprocessing
2. Classifier Discovery
3. Output Results and Cleanup

2.2.2 User Input and Data Preprocessing

Input Parameters and Book-Keeping

Several parameters will be gathered from the GUI by *EvoLearner*:

`$intBits` The accuracy of representation, in number of bits, for nominal attributes.

`$realBits` The accuracy of representation, in number of bits, for continuous attributes.

`$iPop` The initial population size, in number of chromosomes.

`$kPop` The population size to keep each generation for iteration, in number of chromosomes.

`$gPop` The population size to keep each generation for mating, in number of chromosomes.

`$mutRate` The mutation rate, a percentage of the population's total bits.

`$maxGen` The maximum number of generations to iterate.

`$wantFit` The desired level of fitness for classification rules, with fitness defined in section 1.3.

Following the gathering of these parameters, *EvoLearner* will open the schema file and fill two book-keeping data structures which hold all information about the structure of the data file.

1. `@schema` is a two-dimensional (2D) array which holds the descriptor and data type of each attribute listed in the schema file:

$$\begin{pmatrix} [0] & LabelDescriptor & LBL & 1 \\ [1] & Attribute1Descriptor & INT & \texttt{\$intBits} \\ [2] & Attribute2Descriptor & REAL & \texttt{\$realBits} \\ [3] & Attribute3Descriptor & CAT & 4 \end{pmatrix}$$

   Column 3 stores the number of chromosome bits to be used for encoding that particular attribute.

2. `%catHash` is a hash of arrays. It is indexed by each categorical attribute's `@schema` array index, and each array stores the possible values for a categorical attribute:

$$\left(\{2\} \rightarrow [Value1, Value2, Value3, Value4] \right)$$

At this point, *EvoLearner* opens the data file and fills another 2D array, `@data`, with the actual data set. It then begins the process of normalizing and quantizing all numerical values so that they may be easily compared with the decoded genes of the chromosomes.

Normalization

Normalization takes place through two scans of each numerical attribute column:

1. Find the maximum and minimum values of the attribute.

2. Rewrite each normalized value to its respective cell in `@data` by using:

$$V_{norm_i} = \frac{V_i - V_{min}}{V_{max} - V_{min}} \tag{2.1}$$

This process results in all values being relatively scaled to the range $0 \ldots 1$.

Quantization

Quantization divides the range $0 \ldots 1$ into $2^{N_{bits}-1}$ bins, where N_{bits} is equal to the number of bits desired for encoding of that particular attribute (`$intBits` for nominal values, `$realBits` for continuous values). It then sets each value equal to the median of the bin range into which it falls. This minimizes the error associated with quantizing the values, as the encoded value will never be greater than $\frac{binSize}{2}$ from the actual value[HH98]. Quantization takes place in two steps:

1. A 2D array, `@bins`, is filled with the minimum, maximum, and median values of each bin using the following algorithm:

```
for i = 0 to (binDenominator - 1) {
        bins[i][0] = i / binDenominator
        bins[i][1] = (i + 1) / binDenominator
        bins[i][2] = (bins[i][0] + bins[i][1]) / 2
}
```

where `binDenominator` $= 2^{N_{bits}}$, and resulting in the following structure:

$$\begin{pmatrix} [0] & 0 & \frac{1}{2^{N_{bits}}} & \frac{1}{2\left(2^{N_{bits}}\right)} \\ [1] & \frac{1}{2^{N_{bits}}} & \frac{2}{2^{N_{bits}}} & \frac{1+2}{2\left(2^{N_{bits}}\right)} \\ \vdots & \vdots & \vdots & \vdots \\ \left[2^{N_{bits}}-1\right] & \frac{2^{N_{bits}}-1}{2^{N_{bits}}} & 1 & \frac{2\left(2^{N_{bits}}\right)-1}{2\left(2^{N_{bits}}\right)} \end{pmatrix}$$

2. Each value in the column to be quantized is compared with the minimum and maximum values for each bin, and when the correct bin is found the value is rewritten to `@data` as the median of the bin.

Temporary Data Storage

To free up memory, `@data` is then rewritten to a temporary working data file, `evolearn.dat`. This file will be used when evaluating the rules generated by the binary genetic algorithm.

Finding the Chromosome Size

To learn `$chrSize`, the size in bits of the chromosomes, *EvoLearner* simply sums the third column of `@schema`. `REAL` and `INT` genes will be encoded using the number of bits entered by the user. `CAT` genes will be encoded with one bit per possible value of the categorical attribute. A bit string such as 0100 would indicate that the

attribute is equal to value 2. A bit string such as 0011 indicates the disjunction of values 3 and 4, and bit strings such as 0000 and 1111 indicate "don't care" values.

2.2.3 Classifier Discovery

At this point in *EvoLearner's* execution, we are ready to begin use of the binary genetic algorithm. The following process will occur twice, so as to guarantee that a classification rule will be generated for each value of the class label. This execution will be controlled by the value of `$fBit`, whose value will be 1 or 0, and will always be the first bit of every chromosome during a given execution pass.

Initialization

Several steps must be taken before the algorithm can "get off and running":

1. A 2D array of size (`$ipop` × `$chrSize`) will be filled with a random assortment of 1's and 0's, except for the first bit of each row, which will always contain the value of `$fBit`.

2. Using the fitness function (described in section 2.2.5), *EvoLearner* will fill `%fitness`, a hash of the fitness percentages attributed to each chromosome. `%fitness{i}` corresponds to `@population[i]`.

3. Using the sorter function (described in section 2.2.6), *EvoLearner* will sort the chromosomes of `@population` in ascending order by their fitness percentage.

4. A stack-style pop operation will be used to remove (`$ipop` − `$kpop`) rows from `@population`.

Iteration

Finally we've reached the heart of the binary genetic algorithm. The iterative portion actually does the work described in section 1.3. It roughly consists of the following steps:

1. Selection — those chromosomes not having a high enough fitness level are eliminated.

2. Pairing — pairs of chromosomes are selected to mate.

3. Mating — the selected pairs are mated using the crossover operator, and the population is returned to its original size.

4. Mutation — all but the most fit chromosomes are subject to mutation at the rate `$mutRate`.

5. Reranking — the new population is reranked by repeating steps 2 and 3 of Initialization.

The Loop This process continues until either the maximum number of generations has been reached, or until some chromosome's fitness has exceeded the desired fitness level.

1. Again, a stack-style pop operation will be used to remove (`$kPop` − `$gPop`) rows from `@population`.

2. The *rank-weighted random pairing method* will be used to pair chromosomes for mating purposes. It is one of several possible approaches to pairing chromo-

somes. This method assigns a mating probability to each chromosome based on its rank by fitness percentage. It is problem independent in that the probability is not based on the actual value of the fitness. The mating probability is calculated by:

$$P_n = \frac{N_{good} - n + 1}{\sum_{i=1}^{N_{good}} i} \tag{2.2}$$

where $N_{good} =$ `$gPop`. The cumulative probabilities will be stored in an array, `@ranks`. To pair, two random numbers in the range $0 \ldots 1$ are generated. The first chromosome having a cumulative mating probability greater than each generated random number will be a member of the mating pair [HH98]. It is important to notice that chromosomes **can be mated to themselves!** The number of mating pairs selected will be equal to $\frac{\texttt{\$kPop} - \texttt{\$gPop}}{2}$. Each mating produces two children. If an odd number of child chromosomes is required to return the population to its original size, only the first child of the final mating pair will be added to the population.

3. The crossover operator will be used to mate the chromosomes selected by the pairing algorithm. The function will take the two parent chromosomes as parameters and use a stack-style push operation to add the two child chromosomes to `@population`. It will select a random array index, `$crossIndex`, and the first child chromosome will receive `$fBit`, followed by bits $1 \ldots$ `$crossIndex` of $Parent_1$, and bits `$crossIndex` $\ldots$ `$chrSize` of $Parent_2$. The second child chromosome will receive `$fBit`, followed by bits $1 \ldots$ `$crossIndex` of $Parent_2$, and bits `$crossIndex` $\ldots$ `$chrSize` of $Parent_1$.

4. The mutation operator will be used to mutate individual bits throughout the entire population, except within the fittest chromosome, as suggested by [HH98]. The mutator will loop through $\lfloor \frac{\$kPop \times \$chrSize \times \$mutRate}{100} \rfloor$ iterations. Each iteration it will generate a random row number, `$row`, and if this row is not the fittest chromosome, it will generate a random column number, `$column`. If this column number is not the first column, it will then "flip" bit `@population[$row][$column]`.

5. Finally, steps 2 and 3 of section 2.2.3 are repeated.

2.2.4 Output Results and Cleanup

At this point *EvoLearner* will display and test the two best classification rules it could find, one for each of the two class labels. These will be followed by a confusion matrix, giving the number of records in the test data set that are correctly and incorrectly classified. Also displayed will be the classification accuracy:

$$classification\ accuracy = \frac{correct\ classifications}{total\ classifications} \times 100\% \tag{2.3}$$

percent classified:

$$percent\ classified = \frac{total\ classifications}{total\ records} \times 100\% \tag{2.4}$$

and absolute accuracy:

$$absolute\ accuracy = \frac{correct\ classifications}{total\ records} \times 100\% \tag{2.5}$$

It will also display the number of generations that have iterated.

Rule Generation At this point it becomes necessary to "decode" a chromosome. Since we potentially have two fundamentally different data types, we have two different decoding procedures:

1. For Nominal and Continuous-Valued Attributes, we must first obtain the quantized value that evolved within the chromosome. It is found using the following formula from [HH98]:

$$P_{quant} = \sum_{m=1}^{N_{bits}} (@gene[m]2^{-m}) + 2^{-N_{bits}} \tag{2.6}$$

2. For Categorical-Valued Attributes, we will require the use of `%catHash` and `@schema`. Suppose we are working with an attribute indexed by `$i` in `@schema`. First, we will check to see if `@gene` contains a string of all 1's or all 0's. If this is the case, we "don't care" what the value of this attribute is, and the rule will contain the string `DONTCARE` for this attribute. If not, if $\$gene[\$j] = 1$ we will concatenate the string containing `$catHash{$i}[$j]` plus the disjunction symbol `||` onto the rule.

Using these decoding procedures, rule generation will take place as follows, stepping through each non-label "gene" of the chromosome:

1. If it is a numerical gene, append the attribute name, followed by = and the decoded value.

2. If it is a categorical gene, append the attribute name, followed by = and the disjunction created from the decoding.

3. If we "don't care" what the attribute's value is, append the attribute name, followed by = and `DONTCARE`.

After decoding all non-label genes, append `=>` plus the class label, indicated by the first bit of the chromosome.

Testing To test the classification accuracy of the two rules generated, *EvoLearner* will compare the attribute values specified by the rule, or *antecedent*, with the actual values of the test data records. For each match, it will compare the class label specified by the rule, or *consequent*, with the actual label of the matching record. For each match of both antecedent and consequent, a counter of correct values for the appropriate class will be incremented, as well as a counter of total values for that class. For each match of the antecedent followed by a mismatch of the consequent, a counter of incorrect values for the appropriate class will be incremented, as well as a counter of total values for the opposite class. For those data records that do not match the antecedent of either rules, a counter of "unknown" predictions for

that class will be incremented. The accuracy statistics will then be calculated as described above.

Cleanup At this point, *EvoLearner* will offer to write the displayed results to `evolearn.results`, and then delete `evolearn.dat`.

2.2.5 Fitness Function

The fitness function is the "bottle-neck" of the genetic algorithm, with complexity $\mathcal{O}(mn)$ for m training-set records and n chromosomes. It follows these steps:

1. Decode the chromosome into an array, `@decoded`, using the decoding procedures noted in section 2.2.4.

2. Open the temporary data file, `evolearn.dat`, and step through the records:

   - Compare the attribute values from each record to the decoded rule antecedent.
   - If a match is found, increment `$confidenceDenominator`.
   - If the record also matches the decoded rule consequent, increment `$supportNumerator` and `$confidenceNumerator`.
   - Increment `$supportDenominator`.

3. Close the temporary data file.

4. Calculate support:

$$\$\texttt{support} = \frac{\$\texttt{supportNumerator}}{\$\texttt{supportDenominator}} \tag{2.7}$$

   and confidence:

$$\$\texttt{confidence} = \frac{\$\texttt{confidenceNumerator}}{\$\texttt{confidenceDenominator}} \tag{2.8}$$

5. Return \$support $\times$ \$confidence.

2.2.6 Sorter Function

The sorter function is used to place the chromosomes of `@population` in ascending order. This ordering allows some useful operations, such as the stack-style push and pop, to be used for manipulating the chromosomes more easily.

1. Sort the keys of `%fitness` in ascending order by their corresponding fitness values, and store in `@chromoKeys`.

2. Create a 2D array, `@tempPop`, for temporary sorting work.

3. Loop through the chromosomes, setting `$tempPop[$i]` equal to `@{$population[$chromoKeys[$i]]}`.

4. Set `@population` equal to `@tempPop`.

5. Destroy `@tempPop`.

2.3 *NoiseMaker* Design Specification

NoiseMaker will be a research utility for the purposes of noise studies with classification algorithms, and will be implemented in concert with *EvoLearner*. As such, its capabilities as far as data types and input file requirements are exactly the same as those for *EvoLearner*, and can be reviewed in section 2.2.

To best support the many differing types of real-world noise conditions described in section 1.4, *NoiseMaker* will support three types of user-controls:

1. *Attribute-Wise Noise Probability*, the probability that a given attribute contains noise.

2. *Value-Wise Noise Probability*, the probability that a given value within a noise-containing attribute itself contains noise.

3. *Noise Magnitude*, the magnitude of noise within a given numerical value, with 2 possibilities:

    - *Uniform Scaling Factor*, where each random number $(0 \ldots 1)$ generated for the purpose of adding noise to a value is multiplied by a uniform scaling factor, resulting in relatively consistent noise addition.
    - *Random Scaling Factor*, where each random number $(0 \ldots 1)$ generated for the purpose of adding noise to a value is multiplied by a random scaling factor in some user-specified range, resulting in a wider distribution of noise addition.

NoiseMaker's execution can be divided into two distinct steps:

1. User Input

2. Noise Addition and Output

2.3.1 User Input

The parameters gathered by *NoiseMaker* from the GUI are highly dependent and dynamic. Two static data structures, however, are the `@schema` and `%catHash` structures of *EvoLearner*, which can be reviewed in section 2.2.2.

Upon filling those book-keeping data structures, NoiseMaker will generate a table within the GUI where by the user can specify the Attribute-Wise Noise Probability for each attribute within the data file. These values will be stored in `@attProb`.

Also within the GUI will be a field for the global Value-Wise Noise Probability, `$pctNoise`. Future implementations will include the ability to specify an individual Value-Wise Noise Probability for each attribute.

The final user input involves the noise magnitude for numeric data types. Nominal (`INT`) and continuous (`REAL`) data will have separate parameters:

`$IntU_R` A boolean value indicating whether to use uniform (1) or random (0) scaling factors for nominal data.

`$IntUFactor` If using uniform scaling factors, the uniform scaling factor for nominal data.

`@IntRange` If using random scaling factors, the range from which to generate scaling factors for nominal data.

`$RealU_R` A boolean value indicating whether to use uniform (1) or random (0) scaling factors for continuous data.

`$RealUFactor` If using uniform scaling factors, the uniform scaling factor for continuous data.

`@RealRange` If using random scaling factors, the range from which to generate scaling factors for continuous data.

2.3.2 Noise Addition and Output

After gathering the input parameters, *NoiseMaker's* first task will be to decide which attributes will actually be considered for noise addition. It will loop through `@attProb`, generating random numbers and comparing them to the stored probabilities. If `rand` < `$attProb[$i]`, then *NoiseMaker* will add `$i` onto a new array, `@attGen`, which will contain the list of attribute indexes to consider for noise addition.

At this point, *NoiseMaker* will open the data file for reading. It will then enter the following loop:

```
while (<INFILE>) {
        split input line on \t into @record;

        for ($i = 0; $i < $#attGen; $i++) {
                if (rand < $PctNoise) {
                        if ($schema[$attGen[$i]] == 0) {
                                if ($IntU_R) {
                                        UniformScaleInt($attGen[$i]);
                                }
                                else {
                                        RandomScaleInt($attGen[$i]);
                                }
                        elsif ($schema[$attGen[$i] == 1) {
                                if ($RealU_R) {
                                        UniformScaleReal($attGen[$i]);
                                }
                                else {
                                        RandomScaleReal($attGen[$i]);
                                }
                        else {
                                CategoricalNoise($attGen[$i]);
                        }
                }
        }

        print OUTFILE "line just generated";
}
```

The functions `UniformScaleInt`, `RandomScaleInt`, `UniformScaleReal`, `RandomScaleReal`, and `CategoricalNoise` will be described in the following sub-sections. At the conclusion of this loop, *NoiseMaker* has completed its execution, and the output is stored in a file sharing the same first name as the schema and data files, with the extension `.nm`.

Function `UniformScaleInt`

This function performs noise addition to an `INT` value stored at `$record[$attGen[$i]]`, by adding the term $\lfloor \texttt{rand} \times \$\texttt{IntUFactor} \rfloor$.

Function `RandomScaleInt`

This function performs noise addition to an `INT` value stored at `$record[$attGen[$i]]`, through the use of a random scaling factor, `$rFact`. First it calculates `$rFact`:

$$\$\texttt{rFact} = (\texttt{rand} \times \$\texttt{IntRange}[0]) - \$\texttt{IntRange}[1] \tag{2.9}$$

It then adds the term $\lfloor \texttt{rand} \times \$\texttt{rFact} \rfloor$ to `$record[$attGen[$i]]`.

Function `UniformScaleReal`

This function performs noise addition to a `REAL` value stored at `$record[$attGen[$i]]`, by adding the term $\texttt{rand} \times \$\texttt{RealUFactor}$.

Function `RandomScaleReal`

This function performs noise addition to a `REAL` value stored at `$record[$attGen[$i]]`, through the use of a random scaling factor, `$rFact`. First it calculates `$rFact`:

$$\$\texttt{rFact} = (\texttt{rand} \times \$\texttt{RealRange}[0]) - \$\texttt{RealRange}[1] \tag{2.10}$$

It then adds the term $\texttt{rand} \times \$\texttt{rFact}$ to `$record[$attGen[$i]]`.

Function `CategoricalNoise`

This function performs noise addition to a `CAT` value stored at `$record[$attGen[$i]]`.
It first selects a random index, `$j`, such that $\$\texttt{catHash}\{\$\texttt{attGen}[\$\texttt{i}]\}[\$\texttt{j}] \neq \$\texttt{record}[\$\texttt{attGen}[\$\texttt{i}]]$.
It then writes `$catHash{$attGen[$i]}[$j]` to `$record[$attGen[$i]]`.

2.4 Experimental Design

2.4.1 Algorithmic Parameters

During the testing phase of *EvoLearner*, several combinations of the input parameters were applied, and the following set was decided upon as experimental constants (Table 1):

Table 1 – *EvoLearner* Parameters

Parameter	Value
`$intBits`	3
`$realBits`	3
`$iPop`	500
`$kPop`	20
`$gPop`	10
`$mutRate`	2%
`$maxGen`	50
`$wantFit`	1

The choices of three bits for both `INT` and `REAL` representation were consistent with [HH98]'s binary genetic algorithm example. [HH98] further suggests setting `$iPop` $\gg$ `$kPop` so as to begin with a substantial subset of the hypothesis space. 500 and 20 were chosen to fit this suggestion, and values greater than these did not significantly affect performance. The choice of `$gPop` $= .5 \times$ `$kPop` is common within the literature, as is a mutation rate between one and five percent. Fifty

generations were adequate to achieve good performance, and significant favorable evolution rarely occurred beyond this level. As the fitness levels were derived from support and confidence levels, both domain and data dependent values, no cap was put on this value, and all experiments ran the entire fifty generations.

2.4.2 Test Data Sets

Data sets were retrieved from The University of California-Irvine Machine Learning Repository, located at *http://www.ics.uci.edu/ mlearn/MLRepository.html*. The following data sets were chosen:

House Votes 1984 Categorical Congressional voting data from 1984, used to classify United States Representatives by party affiliation.

Wisconsin Breast Cancer Continuously-valued, summarized morphological data used to classify breast cancer tumors as benign or malignant.

Agaricus lepiota Categorical morphological data used to classify mushrooms as either poisonous or edible.

Monks Artificially generated data commonly used to benchmark machine learning algorithms.

2.4.3 Experiments

Initial performance trials were performed using House Votes 1984. Ten replicate trials were run, using 10 random holdout samples of approximately 67% training

and 33% test data. These trials were compared with the results of the WEKA C4.5 Decision Tree algorithm, using the same holdout method. These initial performance trials were followed up with similar trials conducted using the Wisconsin Breast Cancer and *Agaricus lepiota* data sets. Following performance problems with both latter data sets, a final performance trial was conducted with the Monks data set, which although artificial, has similar characteristics to House Votes 1984. The relatively favorable results retrieved somewhat validate the algorithm's excellent performance in classifying House Votes 1984, and the decision was made to continue with noise experiments on both of these data sets.

Two sets of noise experiments were performed, simulating the two noise conditions described in section 1.4. In the first set of experiments, intended to simulate the multiple-attribute noise condition, ten sets of ten replicate trials were run, starting with 0% noise, and increasing noise by 5% in all attributes at the beginning of each set. These trials were compared with the results of WEKA C4.5 under the same conditions. The second set of experiments began with ten trials of WEKA C4.5, and the attribute most often picked as the root of the decision tree was chosen as the single noise containing attribute. Noise was added to the data set as in the first experiment, except only to this single attribute. The results of both *EvoLearner* and WEKA C4.5 were again compared.

Chapter 3

Results

3.1 House Votes 1984

Under the multiple-attribute noise condition, the mean classification accuracy of *EvoLearner* exhibited a slow, steady decline, while the mean percentage of records that *EvoLearner* classified, as well as mean absolute accuracy, showed a much steeper decline (Figure 1). The low classification percentage can be attributed to the fact that *EvoLearner* only generates one rule for each class, and has nothing to predict whatsoever about those rules that do not match either of the rules' antecedents.

In comparing *EvoLearner* with WEKA C4.5, it only makes sense to compare the two algorithms' mean classification accuracies, as WEKA C4.5 always achieved 100% classification, resulting in equal mean classification and absolute accuracies. This result can be attributed to the fact that multiple classification rules can be extracted from a decision tree, so WEKA C4.5 had the benefit of more rules than did *EvoLearner*. However, we can compare how well they did on those records

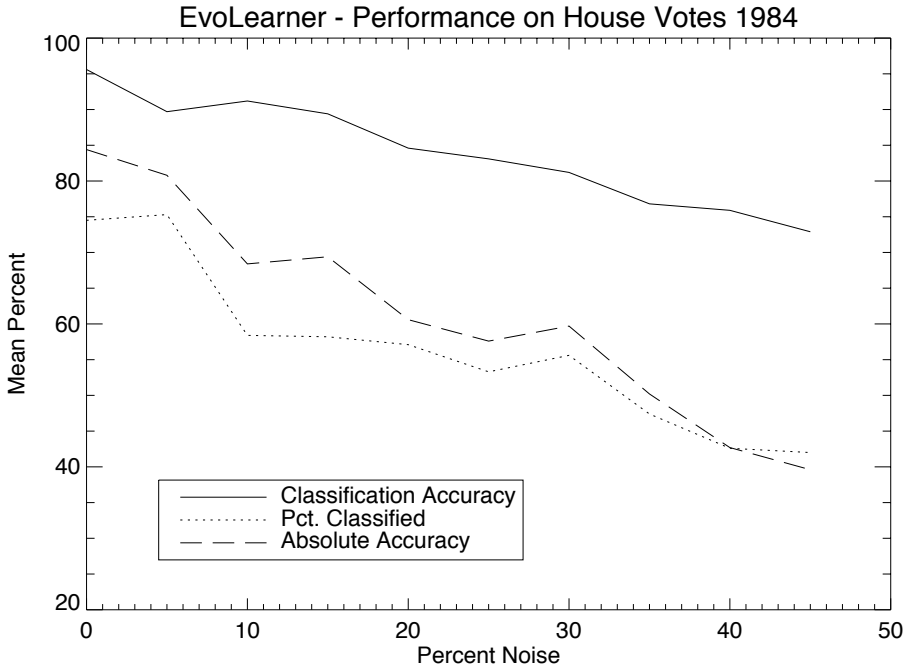

Figure 1 – EvoLearner Performance
Noise in Multiple Attributes
House Votes 1984

that they were able to classify. The results show that in the 0...20% noise range, both algorithms performed equally rather well. Beyond 20% noise, both algorithms exhibit a rather erratic, steep decline in accuracy, with WEKA C4.5 performing slightly better overall (Figure 2).

As described earlier, to simulate the single-attribute noise condition, the root of WEKA C4.5's decision tree inducted over a clean data set was chosen. Over ten trials, the root attribute was "Physician Fee Freeze" 100% of the time. So, noise was introduced into only this attribute. The mean classification accuracy for *EvoLearner* fell approximately 5% after the first injection of noise, and then hovered around the same percentage from that point onward. Again, mean percent classification and

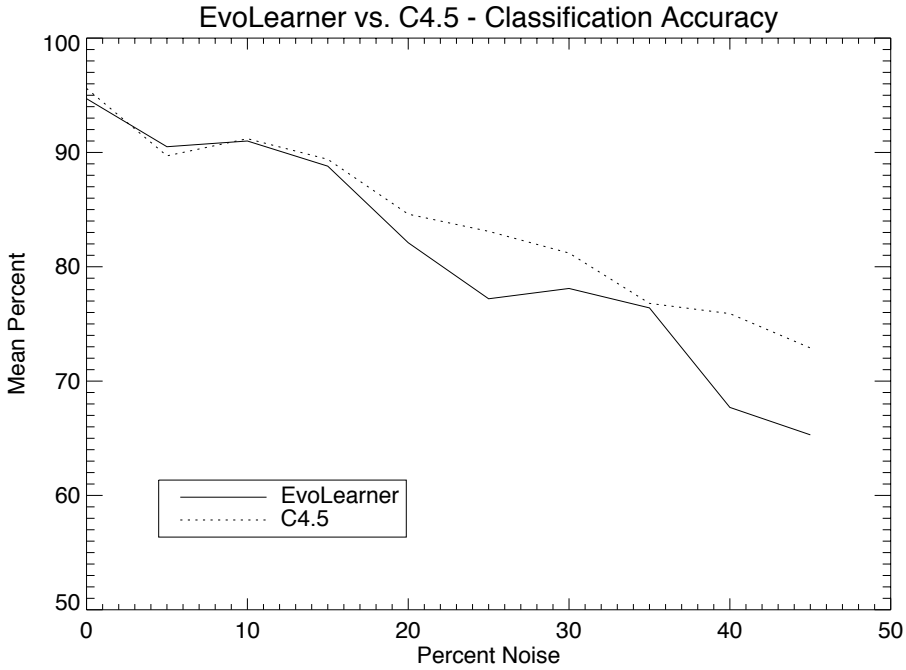

Figure 2 – EvoLearner vs. C4.5
Noise in Multiple Attributes
House Votes 1984

absolute accuracy were lower, and somewhat more erratic. However, they also tended to hover around the same percentage throughout the experiment (Figure 3). Comparing the two algorithms under these conditions showed that both algorithms performed relatively the same, with *EvoLearner* possessing a slight advantage in mean classification accuracy over WEKA C4.5 (Figure 4).

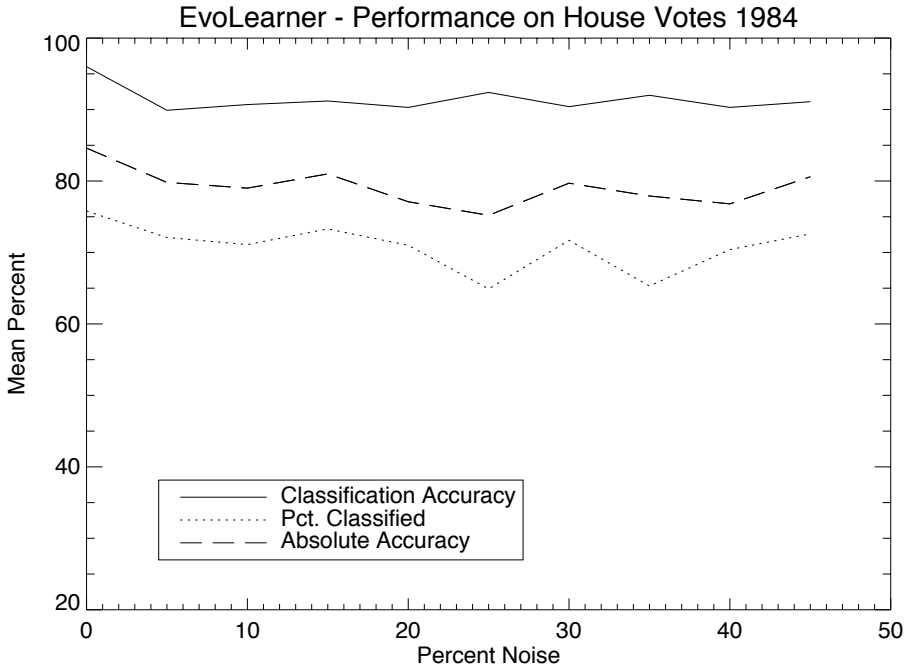

Figure 3 – EvoLearner Performance
Noise in Single Attribute - Physician Fee Freeze
House Votes 1984

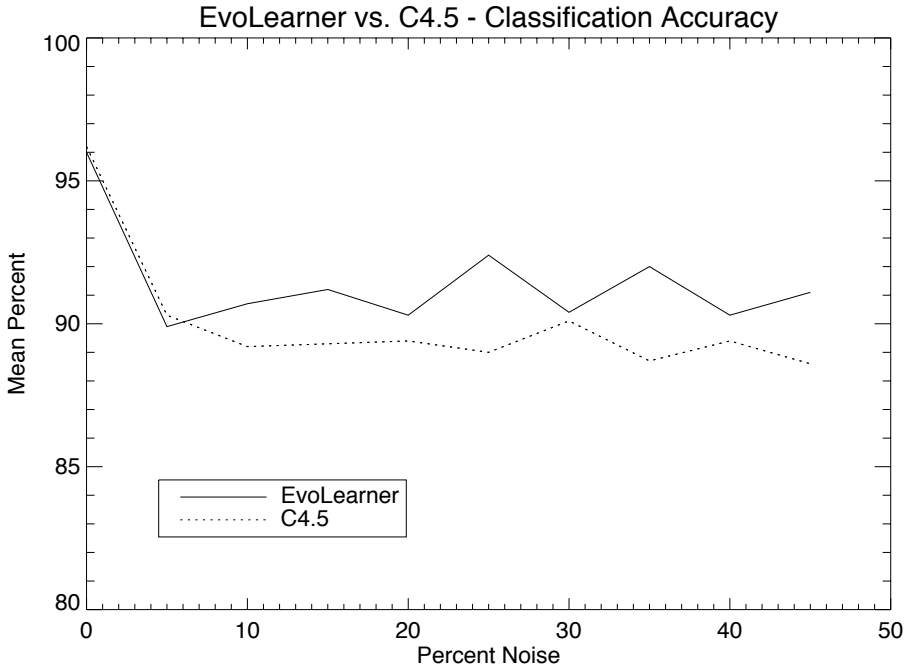

Figure 4 - EvoLearner vs. C4.5
Noise in Single Attribute - Physician Fee Freeze
House Votes 1984

3.2 Wisconsin Breast Cancer

The results for the Wisconsin Breast Cancer data set are summarized in Table 2:

Table 2 – *EvoLearner* – Performance on Wisconsin Breast Cancer

Classification Accuracy	0%
Percent Classified	0%
Absolute Accuracy	0%

Needless to say, these results were alarming. Preliminary indications were that errors were present within the continuous attribute handling mechanisms of *EvoLearner*. Some errors were indeed present, but, upon correcting all visible coding errors, no performance increase was felt. Scanning a debug trace indicated that rules were indeed being generated correctly, but that no records fully matched a rule antecedent.

3.3 *Agaricus lepiota*

Duplicate results were recorded when testing the *Agaricus lepiota* data set (Table 3):

Table 3 – *EvoLearner* – Performance on *Agaricus lepiota*

Classification Accuracy	0%
Percent Classified	0%
Absolute Accuracy	0%

These results were not only alarming, but confusing, given *EvoLearner*'s satisfactory performance on House Votes 1984. Both data sets consist of purely categorical attributes; however, *Agaricus lepiota* is somewhat larger in number of attributes (22 VS. 16) and much larger in values per attribute (10 max, 6 avg per VS. 3 per).

3.4 Monks

Given the poor performance of *EvoLearner* on both of the previous two data sets, the Monks data set was chosen to aid in validating the results obtained from House Votes 1984. *EvoLearner* did not perform nearly as well on this data set, never surpassing 78% in classification accuracy; however, *EvoLearner* was nearly always able to classify 100% of the test records. These results were again confusing, given that WEKA C4.5 classified the same test set at 95% accuracy. Examination of the rules generated by *EvoLearner* revealed the following rule set in 99% of the performance trials:

$$(Jacket\,Color = \text{“}red''\text{)} \Rightarrow (Answer = \text{“}yes''\text{)} \tag{3.1}$$

$$(Jacket\,Color = \text{“}yellow'' \vee \text{“}blue'' \vee \text{“}green''\text{)} \Rightarrow (Answer = \text{“}no''\text{)} \tag{3.2}$$

This result prompted closer examination of the data set. Statistical analysis revealed that zero records included the combination of $(Jacket\,Color = \text{“}red''\text{)}$ and $(Answer = \text{“}no''\text{)}$. However, those records labeled “yes” possessed a roughly equal proportion of all possible values for Jacket Color. These facts more than explain the results. Once again, C4.5 is the beneficiary of more rules than *EvoLearner*, allowing it better performance. *EvoLearner* is choosing the best rule, but the aid of auxiliary rules would likely boost its performance.

Given that *EvoLearner* did indeed appear to be functioning correctly, noise trials were completed. Under the multiple-attribute noise condition, the results are very

interesting. WEKA C4.5's mean classification accuracy fell from 95% to 55% as noise increased from 0% to 45%; however, over the same noise interval, *EvoLearner*'s mean classification accuracy fell from 76% to 55%, a much smaller and slower decent (Figure 5). While the initial accuracies are not really close enough for a conclusive comparison, these results do seem to indicate that *EvoLearner* was less affected by the injected noise than was WEKA C4.5.

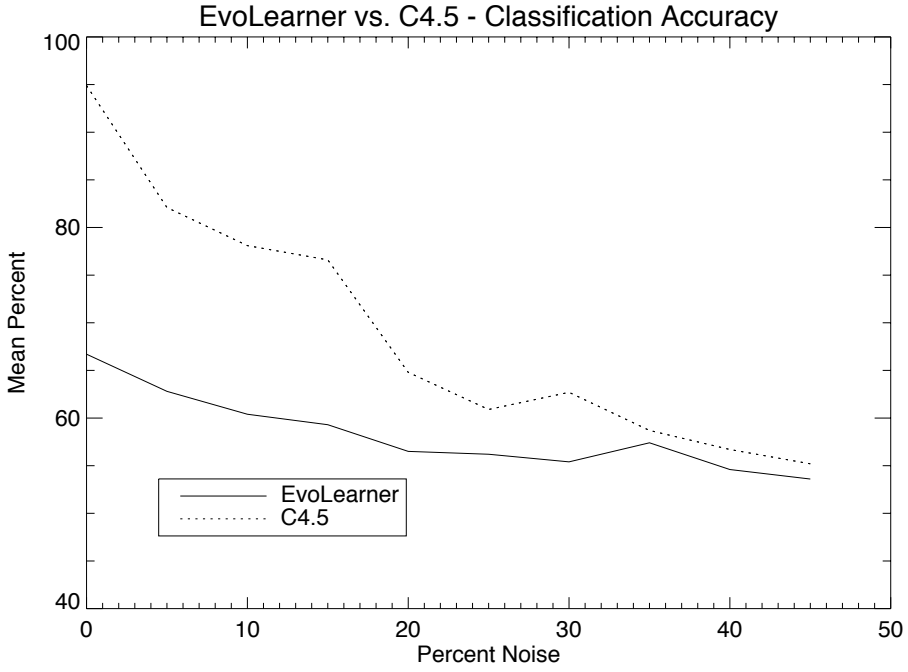

Figure 5 - EvoLearner vs. C4.5
Noise in Multiple Attributes
Monks

To simulate the single-attribute noise condition, once again ten trials were run with WEKA C4.5. Nine of ten trials found "Jacket Color" as the root attribute of the decision tree, which was not surprising given the previous analysis of the data

set. Noise was injected into only this attribute, and again the results are interesting. WEKA C4.5 exhibits a very erratic decent in classification accuracy from near 100% to 85%, while *EvoLearner* exhibits a pattern very close to that of its performance on the single-attribute noise condition in House Votes 1984 (Figures 4 and 6).

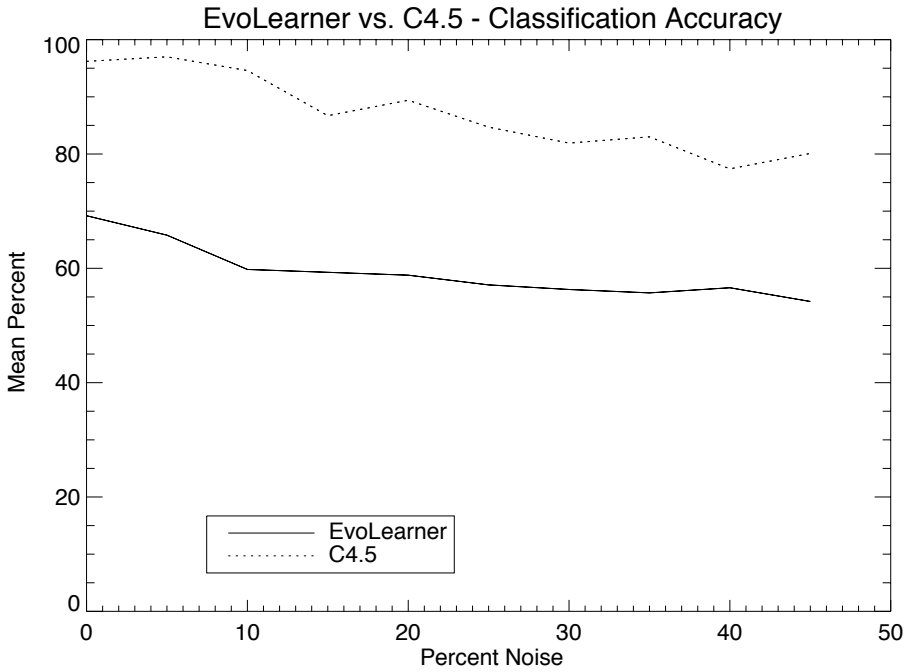

Figure 6 - EvoLearner vs. C4.5
Noise in Single Attribute - Jacket Color
Monks

Again ignoring the difference in initial accuracy, it seems that *EvoLearner* was less affected by the injected noise than was WEKA C4.5.

Chapter 4

Conclusions and Future Challenges

4.1 Conclusions

Four main conclusions were drawn from this study:

1. More classification rules are needed to boost *EvoLearner*'s performance.

2. The present method of encoding categorical attributes is not scalable to large hypothesis spaces.

3. The present method of encoding numerical attributes is unreasonably strict.

4. More research is required to determine just how tolerant the genetic algorithm is of noise. The two successful data sets produced somewhat opposing results; however, its tolerance is at least comparable to the C4.5 decision tree.

EvoLearner suffered in both classification accuracy and in the amount of records it was able to classify. This study has made it painfully obvious that one rule per

class is not enough to guarantee a full, accurate classification of all data objects. Its excellent performance on the clean House Votes 1984 data set results from the fact that the two rules it chose were sufficient for classifying the majority of Congressmen: those who vote strictly down party lines. However, the rules said little about the moderates. Records representing those Congressmen would unlikely match the rule antecedents learned by *EvoLearner*, resulting in an "unknown" prediction. The same can be said of *EvoLearner*'s performance on the Monks data set. WEKA C4.5 often achieved 100% classification accuracy, but only because it had the benefit of additional rules. Had *EvoLearner* possessed a rule that somehow constrained "Body Shape" or "Head Shape," it may have faired better.

Two possible solutions for the rule problem are:

1. Allow a range of fitness values to survive the final cut of classification rules;
2. Perform random restarts to explore more of the hypothesis space, and keep the maximum fitness rule from each execution.

Both these solutions seem adequate in dealing with the amount of rules generated; however, how should the rule family be tested? One possible solution is to use the first rule that a record "fires" (i.e. matches the antecedent) for prediction of that record's class. Consider this situation. Two rules exist in the final rule family which a given record can fire. One of these rules correctly predicts the record's class, while the other predicts incorrectly. Suppose that the incorrect rule is the first fired, resulting in an inaccurate prediction. What does this say about the rule family's accuracy? Perhaps the answer is to inspect each rule until either a correct

prediction is found, or until all rules have been exhausted. This solution seems inefficent given a large family of rules. Further, what if the correctly predicting rule only aids in classifying that single, possibly outlier, record? Should it be saved for later prediction? These are all questions that should be pursued in the future.

Evolearner did not perform well in classifying the records of *Agaricus lepiota*. However, consider the amount of possible chromosomes for this classification problem. Each gene represents one attribute, and has $2^{possible\,values}$ bits. Combinatorics teaches us that the number of possible chromosomes is simply the product of the possible gene configurations. For *Agaricus lepiota*, this number is $2^{124} \approx 2.13 \times 10^{37}$, an impossibly huge number! Contrast this with the hypothesis space of House Votes 1984, with a size of $2^{48} \approx 2.81 \times 10^{14}$, a much more reasonable number. The growth of hypothesis space size is presented in Figure 7.

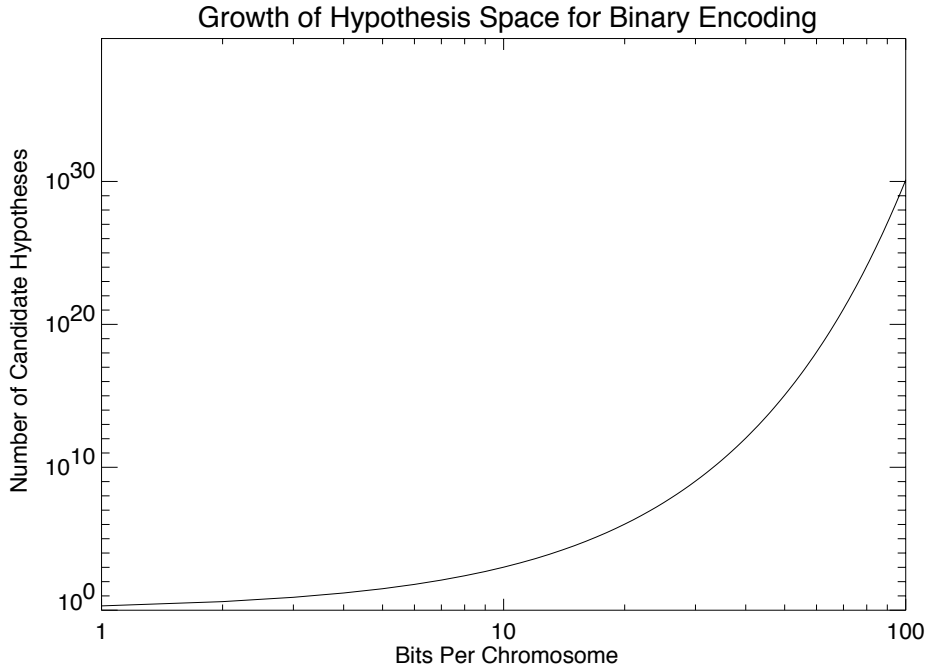

Figure 7 - Growth of Hypothesis Space Size for Binary Encoding

Just like the genetic and evolutionary processes it models, the binary genetic algorithm is governed by randomness and the laws of probability. Strong performance of the genetic algorithm is contingent upon the presence of some positive fitness values in the population. This initial population is generated completely at random. Given the sheer size of *Agaricus lepiota*'s hypothesis space, the probability is almost nil that chromosomes with positive fitness values will be generated, even multiplying this study's initial population size by a factor of ten. Further, as the initial population size grows, we begin to feel the effects of the fitness function's $\mathcal{O}(mn)$ complexity.

The reason for the genetic algorithm's dependence on the initial population including chromosomes with positive fitness values is purely an implementation issue. *EvoLearner* operates by sorting the chromosomes by fitness value, and then trimming the worst n chromosomes. If all chromosomes have a fitness of 0, they will be sorted in an arbitrary order. Perhaps a chromosome exists that is one bit mutation from positive fitness. Because of this random ordering, there is no way to guarantee that chromosome's survival. Another technique for "natural selection" is *thresholding*. With thresholding, all chromosomes having a fitness less than a certain threshold will be removed from the population. If all chromosomes have a fitness of 0, all will be trimmed. If we only allow a certain number of chromosomes to be trimmed, we are right back where we started. Obviously then we must consider some new form of encoding for categorical attributes so that we have a higher probability of obtaining chromosomes with positive fitness values in the initial population, again a question for further research.

Encoding was also a huge problem with numeric attributes. Similar problems existed with the size of the hypothesis space. As the number of bits specified for encoding numeric attributes increased, the hypothesis space again grew exponentially. So, the greater the accuracy of value representation, the lesser the probability of chromosomes with positive fitness values, obviously the opposite of intuition. However, this phenomenon was not the primary problem. The encoding method specified by [HH98] was primarily intended for the purpose of optimizing mathematical functions. It proved far too strict for classification purposes. At the heart of the problem is the fact that all possible genes for a continuous attribute represent specific values. Unlike the encoding for categorical attributes, "don't care" configurations are not present. As a result, for a purely numeric data record to fire a rule, it must match some specific value for each attribute. The method implies that all attributes are relevant to a given record's classification, something we know not to be true for virtually any domain.

A possible solution to the numerical encoding problem is to discretize values, and then use an encoding similar to that used for categorical attributes. Following normalization, some histogram analysis could be performed, resulting in "binned" data. Whether to use an equiwidth or equidepth binning method probably depends on the data set. Encoding could then be performed by assigning each bin a bit in the gene. This encoding would allow for values to fall into more than one bin, allowing a more general range of values to be specified within a rule for a given attribute. Again, strings completely composed of 1's or 0's would represent "don't care" values, allowing irrelevant attributes to be excluded from classification rules.

The problem of hypothesis space size is, like the original numerical encoding, only a problem if we make it one. Unlike the constraints imposed by categorical data, we have the ability to specify the number of bins to be used, thus specifying the size of the hypothesis space.

As the original purpose of this research was to test the binary genetic algorithm in the face of noisy data, something should be said about its noise tolerance. Unfortunately, due to the problems analyzed above, a great enough breadth of testing was not accomplished so as to draw a decisive conclusion. In classifying House Votes 1984, *EvoLearner* performed almost equally as well as WEKA C4.5 under both the multiple-attribute and single-attribute noise conditions. Given that noisy values are probably seldom in excess of 20%, both algorithms' performances in this range is encouraging for the multiple attribute noise condition. Both algorithms performed exceptionally well under the single-attribute noise condition, however this may only be a symptom of the House Votes 1984 data set. As was previously mentioned, the majority of Congressmen voted solely along party lines, and the majority of bills present in the data set were extremely partisan. Unfortunately, no real (non-artificial) data sets were available during this study that were comparable to House Votes 1984 in complexity. *EvoLearner*'s response to noise was encouraging; however, the low starting classification accuracy served to dampen these favorable results. What we can say is that given what we know about *EvoLearner*'s need for additional classification rules, both algorithms seem to perform on a relatively equivalent level in the face of noise. Only additional research can provide more definitive results.

4.2 Future Challenges

Two obvious future research challenges were posed in the previous section:

1. How to test a family of rules for classification accuracy;

2. How to perform encoding so as to achieve a better sampling of the hypothesis space in the initial population.

While no easy solution is evident for the first problem, the second could lead in an interesting direction. Why not, instead of altering the encoding method, simply reduce the size of the hypothesis space? This idea may seem impossible at first glance. Is not the number of possible values for a given attribute fixed by the data domain? Consider the idea of *concept hierarchies*. Concept hierarchies provide some mapping from low-level concepts to higher-level, more general concepts [HK01]. Many concept hierarchies are already implicit within a given data domain. Consider the hierarchy of a person's location. We can define a total order for this attribute:

$$\textit{street address} \rightarrow \textit{city} \rightarrow \textit{province or state} \rightarrow \textit{country} \tag{4.1}$$

Other concept hierarchies could be partial orders, such as one we might imagine for time [HK01]. Concept hierarchies could easily reduce the number of possible values for a given attribute, thus reducing the size of the hypothesis space. Providing some method for data preprocessing by which the use of predefined concept hierarchies can be used to reduce the complexity of a given data set could easily improve upon the classification accuracy of *EvoLearner*, while perhaps alleviating

the need to alter the encoding method. However, some data sets may contain attributes not possessing an apparent concept hierarchy; in fact, some attributes may have none at all. For useful research in this direction to continue, the cooperation of domain experts will be required to supply the required concept hierarchies. Until then, encoding must be explored.

On a closing note, this paper began with the remark that the amount of data archived across multiple distinct domains is growing at an exponential rate. Unfortunately, the amount of freely available data for algorithmic research is not growing at a proportional rate. Increased participation by scientists in such projects as the UCI Machine Learning Repository can only serve to speed useful research into data mining algorithms. Improvements derived from this research can only serve to aid the scientists who first provided the needed data. The result is a win-win situation for all.

Bibliography

[HH98] Randy L. Haupt and Sue Ellen Haupt. *Practical Genetic Algorithms*. Wiley, New York, 1998.

[HK01] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Academic Press, San Diego, 2001.

[Mit96] Melanie Mitchell. *An Introdution to Genetic Algorithms*. MIT Press, Cambridge, 1996.

[Mit97] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, Boston, 1997.

[Qui93] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, CA, 1993.

[WF00] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, San Mateo, CA, 2000.